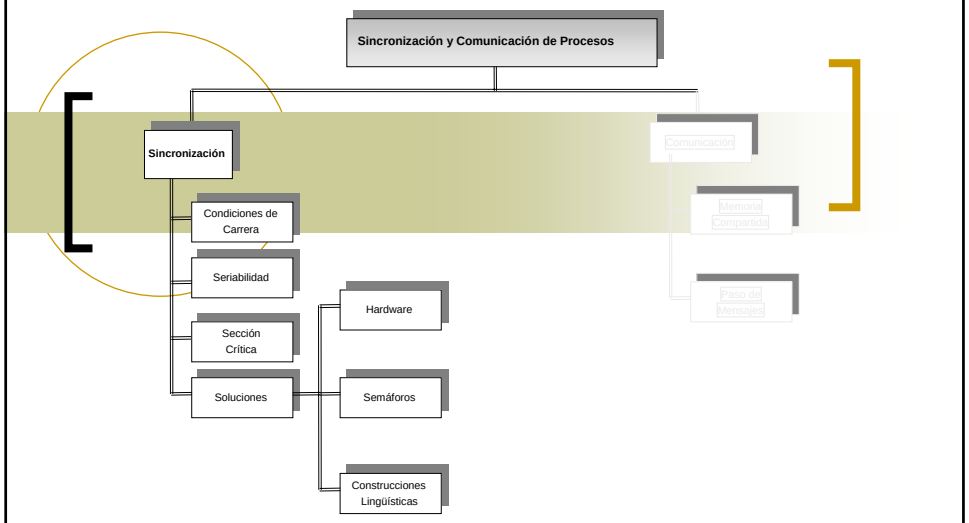


1

Sincronización y Comunicación (I)

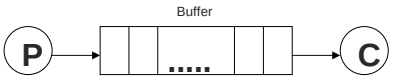
- Sistema multiprogramado ⇒ procesos concurrentes cooperantes
- Datos de un proceso P_i deben ser conocidos por el proceso P_j
- Formas de comunicación
  - Mediante memoria compartida
    - Los procesos se ejecutan en el mismo espacio de direcciones (threads) o tienen acceso a memoria común.
    - Se comunican mediante variables globales
  - Mediante paso de mensajes
    - Los procesos se pueden ejecutar en espacios de direcciones independientes
    - Se comunican mediante mensajes
    - La infraestructura de la comunicación la ofrece el SO (*ports, pipes, ...*)

Sincronización y Comunicación (II)

- Es necesario sincronizar el acceso a zonas de memoria compartida para mantener la integridad de los datos
- Un dato puede perder la integridad cuando varios procesos lo modifican concurrentemente
- *Seriabilidad*: la ejecución paralela de un conjunto de instrucciones debe ser equivalente a la de una de las posibles alternativas secuenciales
- Si no se cumple la seriabilidad se dice que se ha producido una ***condición de carrera***

Ejemplo: Productor - Consumidor (I)

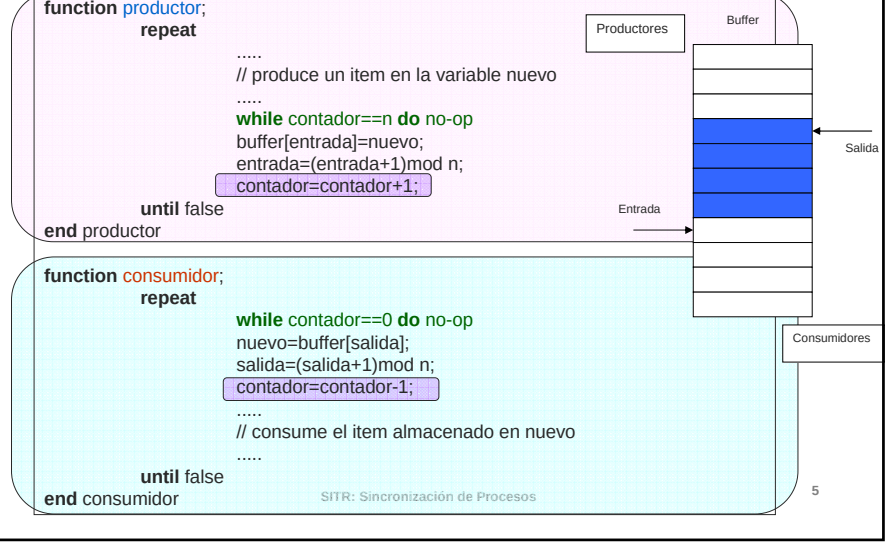
- Se plantea la necesidad de comunicar la información generada por un grupo de procesos (productores) a otro grupo que necesitan de ella (consumidores)
- En la comunicación se utiliza una zona de memoria de tamaño limitado (buffer) que almacena los datos producidos y no consumidos



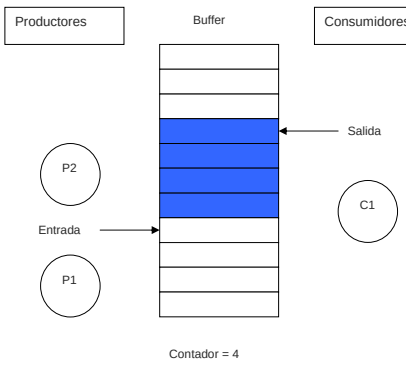
- El problema consiste en que varios procesos acceden concurrentemente a una zona de datos compartidos y se pueden dar condiciones de carrera

```
// Problema productor - consumidor
type item=.....;
var         buffer:array[0..n-1] of item
entrada, salida: 0..n-1;
contador: 0..n
entrada=0; salida=0; contador=0;
function productor;
        repeat
                .....
                // produce un item en la variable nuevo
                .....
                while contador==n do no-op
                buffer[entrada]=nuevo;
                entrada=(entrada+1)mod n;
                contador=contador+1;
        until false
end productor

function consumidor;
        repeat
                while contador==0 do no-op
                nuevo=buffer[salida];
                salida=(salida+1)mod n;
                contador=contador-1;
                .....
                // consume el item almacenado en nuevo
                .....
        until false
end consumidor
```



Ejemplo: Productor - Consumidor (II)



- contador = contador +1

```
mov    contador, reg1
inc    reg1
mov    reg1, contador
```

- contador = contador - 1

```
mov    contador, reg2
dec    reg2
mov    reg2, contador
```

Ejemplo: Productor - Consumidor (III)

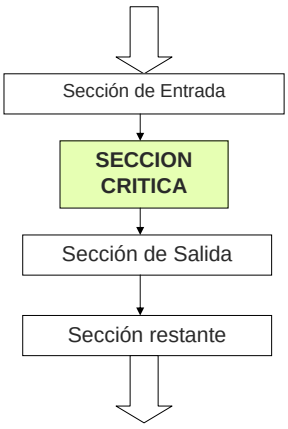
- Supongamos que se da la siguiente ejecución

Tiempo	Proceso	Operación	reg1	reg2	contador
t0	productor	mov contador, reg1	5	?	5
t1	productor	inc reg1	6	?	5
t2	consumidor	mov contador, reg2	6	5	5
t3	consumidor	dec reg2	6	4	5
t4	productor	mov reg1, contador	6	4	6
t5	consumidor	mov reg2, contador	6	4	4

- La variable *contador* es utilizada por el proceso consumidor antes de que el proceso productor la actualice ⇒ inconsistencia (vale 4 en vez de 5)
- CONCLUSION: este tipo de operaciones deben ser ATÓMICAS

El Problema de la Sección Crítica (I)

- Intenta sistematizar el estudio de la sincronización
- Sean *n* procesos que tienen datos compartidos. Cada proceso tiene un segmento de código llamado ***Sección Crítica*** en el cual se accede a los datos compartidos
- Se pretende que cuando un proceso está en su sección crítica, ningún otro este en la suya propia



El Problema de la Sección Crítica (II)

- Para resolver el problema de la sección crítica hay que definir el código de la sección de entrada y de la sección de salida
- Una solución al problema de la sección crítica debe cumplir las siguientes propiedades
  - *Exclusión mutua*: si P_i ejecuta su sección crítica, ningún otro P_j ($i \neq j$) lo podrá hacer en la suya
  - *Progreso*: Si no hay procesos ejecutando su sección crítica y hay varios que quieren entrar en la suya, la decisión de quien entra la deben tomar aquellos que no se encuentre en su sección restante y en un tiempo finito
  - *Espera limitada*: Un proceso no debe esperar infinitamente a ejecutar su sección crítica
- El problema de la sección crítica se puede resolver son soluciones software o hardware

El Problema de la Sección Crítica (III)

- Soluciones hardware:
  - Inhibición de interrupciones
    - La forma más sencilla y menos eficiente de resolver el problema
    - No se permiten interrupciones durante la ejecución de la sección crítica (así se evitan los cambios de contexto)

```
Inhibir_Interrupciones();
Sección crítica
Habilitar Interrupciones();
```

    - Sólo es útil cuando la sección crítica es pequeña
    - Este método de sincronización sólo es aplicable a nivel del SO (es peligroso que los usuarios accedan al sistema de interrupciones)

Semáforos (I)

- Las soluciones hardware anteriores no son fáciles de generalizar a problemas más complejos
- Semáforo (S): estructura de datos formada por una variable tipo entero y una cola de procesos en espera
- A un semáforo sólo se puede acceder mediante dos operaciones P y V

```
typedef semaforo
{
        int contador;
        cola: lista de procesos;
}
```


Semáforos (II)

- P(S)
  - Decrementa el contador
  - Si el resultado es negativo suspende al proceso en la cola asociada
- V(S)
  - Incrementa el contador
  - Si el resultado es negativo o cero despierta a un proceso que se encuentra en la cola del semáforo

```
S.contador = S.contador-1;
if (contador<0)
{
     Insertar_en_Cola(S.cola,
                 Proceso_en_ejecución);
     Suspender(Proceso_en_ejecución);
}
```

```
S.contador = S.contador+1;
if (contador<=0)
{
          Extraer_de_Cola(S.cola, Proceso);
          Activar(Proceso);
}
```

Semáforos (III)

- Si el contador es mayor o igual que cero, indica el número de procesos que pueden invocar la operación P sin suspenderse
- Si el contador es menor o igual que cero, su valor absoluto indica el número de procesos suspendidos en la cola asociada al semáforo
- P y V (suministradas por el SO) deben realizarse de forma atómica ya que su código es una sección crítica
- El uso de semáforos no supone espera activa por lo que no exige gasto innecesario de CPU

Empleo de Semáforos (I)

- Problema de la sección crítica para n procesos
  - Se tienen n procesos que comparten datos pero sólo uno de puede acceder a ellos (Exclusión Mutua, *mutex*)
  - Se define el semáforo

```
var mutex: semaforo(1) (*El valor inicial del contador es 1*)
```

  - Cada proceso Pi tiene la estructura siguiente

```
.
.
P(mutex);  /*Sección de entrada */
 Sección crítica
V(mutex);  /* Sección de salida */
 Sección restante
.
.
```

Empleo de Semáforos (II)

- Secuenciación de dos secciones de código de procesos diferentes
  - Se pretende ejecutar B en *Pj* después de A en *Pi*
  - La solución consiste en crear un semáforo S con S.contador = 0 y definir las secuencias de *Pi* y *Pj* como sigue

Pi:	Pj:
.....	.....
A	P(S)
V(S)	B
.....	.....

Empleo de Semáforos (III)

- Aparición de interbloqueos
  - Al utilizar semáforos se pueden dar situaciones en que ningún proceso pueda acceder a su sección crítica y todos los procesos que comparten el semáforo estén suspendidos
  - Supongamos que tenemos dos procesos P1 y P2 que comparten dos semáforos S y Q

P1:	P2:
.....	.....
P(S);	P(Q);
P(Q);	P(S);
.....	.....
V(S);	V(Q);
V(Q);	V(S);
.....	.....

```
P2: P(Q)
P1: P(S)
P1: P(Q) (se suspende)
P2: P(S) (se suspende)
```

Empleo de Semáforos (IV)

- Inversión de Prioridad:
  - Es un tipo de interbloqueo entre procesos cuando se utiliza un planificador basado en prioridades fijas
  - Supongamos que tenemos tres procesos P1, P2 y P3 con prioridades (3, 2 y 1) respectivamente. Los procesos P1 y P3 acceden a una sección crítica mediante un semáforo S siendo P3 un proceso largo

```
P1:        P2:        P3:
           P(Q);

.....      .....      .....
P(Q);      V(Q);      P(S);
P(S);
.....      .....      .....
V(S);
V(Q);                 V(S);
.....      .....      .....
```

```
P2: P(Q)-> 0
P1: P(Q)->-1
P3: P(S)-> 0
P2: se activa expulsando a P3 por ser de mayor prioridad
P2: V(Q);
P1: P(S)(se suspende)
P1 queda bloqueado por P2 a pesar de tener menos prioridad
```


Construcciones Lingüísticas

- La utilización incorrecta de los semáforos pueden provocar errores de sincronización
- Para evitar estos errores se han diseñado construcciones lingüísticas de alto nivel que simplifican la programación de aplicaciones concurrentes.
- Construcciones teóricas
  - Monitores: estructuras de alto nivel que facilitan la utilización de esquemas de sincronización
  - Regiones críticas
- Construcciones prácticas
  - Tipos protegidos en ADA'95
  - Objetos de sincronización en JAVA